Unit Testing Activity

SWEN-261
Introduction to Software Engineering

Department of Software Engineering
Rochester Institute of Technology
Your activity for the Unit Testing lesson is to build tests for existing Project components.

- These slides direct you through the process of creating unit tests for your project.
  ✓ Activity actions are highlighted in green with a checkmark.

- But first, these slides provide technical details on:
  1. How to organize test classes using Maven
  2. How to run tests using Maven
  3. How to structure a typical test class
  4. How to use JUnit assertion functions
  5. How to use package-private constants in test code
  6. How to use Mockito mock objects using Dependency Injection
Maven provides a convenient structure for organizing unit test code.

- Put your test classes in a separate source path.
  - The goal is to keep the test source code separate from the production source code.
  - Using Maven that is usually src/test/java.
    - Create this directory if it doesn't already exist.
    - (Optional) Link your IDE to this source path.

- Most IDEs provide wizards for creating unit tests.
  - Make sure the IDE stores the test classes in the proper source path.

- The unit test code examples in these slides are from the Guessing Game.
  - See the Unit Testing topic page for a link to the ZIP file.
Maven will run tests during builds and there is also the test target.

```
[~/workspace/guessing-game] ➔ mvn clean test
[INFO] Scanning for projects...

// SKIPPING SOME Maven OUTPUT

TESTS

Running com.example.appl.GameCenterTest
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.03 sec - in com.example.appl.GameCenterTest

Running com.example.appl.PlayerServicesTest
Tests run: 9, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.831 sec - in com.example.appl.PlayerServicesTest

Running com.example.model.GuessGameTest
Tests run: 11, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 sec - in com.example.model.GuessGameTest

Running com.example.ui.GameViewTest
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.398 sec - in com.example.ui.GameViewTest

Running com.example.ui.GetGameRouteTest
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.414 sec - in com.example.ui.GetGameRouteTest

Running com.example.ui.GetHomeRouteTest
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 sec - in com.example.ui.GetHomeRouteTest

Running com.example.ui.HomeViewTest
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec - in com.example.ui.HomeViewTest

Running com.example.ui.PostGuessRouteTest
Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.014 sec - in com.example.ui.PostGuessRouteTest

Results:
Tests run: 40, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```
Test files are kept in separate directories from the application code but in the same package.

- Name the test class after the *component under test* (CuT) in the same package.
  - *So if CuT is* `com.example.model.MyEntity`
  - *Then test class is:* `com.example.model.MyEntityTest`
  - *Doing so gives the test code package-private access to CuT class.*
JUnit understands several annotations that you must use in your test files.

- Annotate each class with [@Tag] to indicate which architectural tier the class is in.
  - **Use these tags:** Model-tier, Application-tier, UI-tier
  - **You will learn more about the role of these tags in the Code Coverage lesson.**

- Annotate each test method with [@Test].

- **Use [@BeforeEach or @BeforeAll annotated methods for setup before each test or setup done once before all tests**

- **Method annotations [@AfterEach and @AfterAll serve similar clean up tasks after running tests.**
Recall the checklist of types of unit tests.

- Business logic
  - Tests for each path through a method
  - Happy path as well as failure paths
- Constructors and accessors
- Defensive programming checks
  - Validation of method arguments
    - NullPointerException
    - IllegalArgumentException
  - Validation of component state
    - IllegalStateException
- The equals and hashCode methods, as needed
- Exception handling
Here's an example unit test suite for the GuessGame class.

```java
package com.example.model;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;
@Tag("Model-tier")
public class GuessGameTest {
    @Test
    public void ctor_noArg() { ... }
    @Test
    public void ctor_withArg() { ... }
    @Test
    public void ctor_tooBig() { ... }
    @Test
    public void ctor_tooSmall() { ... }
    @Test
    public void isValidGuess() { ... }
    @Test
    public void make_an_invalid_guess_too_small() { ... }
    @Test
    public void make_an_invalid_guess_too_big() { ... }
    @Test
    public void testGamePlay_win_first_try() { ... }
    @Test
    public void testGamePlay_win_second_try() { ... }
    @Test
    public void testGamePlay_win_last_try() { ... }
    @Test
    public void testGamePlay_lose() { ... }
}
```

- **Import JUnit assertion functions**
- **Indicate architectural tier**
- **Test constructors**
- **Test defensive programming checks.**
- **Test business logic.**
Here's an example test method for the GuessGame class.

```java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
@Tag("Model-tier")
public class GuessGameTest {
    @Test
    public void testGamePlay_win_first_try() {
        // Arrange the test scenario & invoke test (on ctor)
        final GuessGame CuT = new GuessGame(NUMBER);

        // Analyze results: game start
        assertTrue(CuT.isGameBeginning());
        assertTrue(CuT.hasMoreGuesses());
        assertEquals(3, CuT.guessesLeft());

        // Invoke test: first guess, correct
        assertEquals(CuT.makeGuess(NUMBER), GuessResult.WON);

        // Analyze results
        assertFalse(CuT.isGameBeginning());
        assertTrue(CuT.hasMoreGuesses());
        assertEquals(2, CuT.guessesLeft());
        assertTrue(CuT.isFinished());
    }

    // Additional tests...
}
```
Here's an example of how to test an expected exception.

- Use the `assertThrows` assertion:

```java
@Tag("Model-tier")
public class GuessGameTest {
    private static final int TOO_BIG = 10;
    @Test
    public void ctor_tooBig() {
        assertThrows(IllegalArgumentException.class, () -> {
            new GuessGame(TOO_BIG);
        }, "GuessGame allowed large number.");
    }
}
```

- Roughly the same as:

```java
@Tag("Model-tier")
public class GuessGameTest {
    private static final int TOO_BIG = 10;
    @Test
    public void ctor_tooBig() {
        try {
            new GuessGame(TOO_BIG);
            fail("GuessGame allowed large number.");
        } catch (IllegalArgumentException e) {
            // success, do nothing
        }
    }
}
```
JUnit has many built-in assertions you can use.

- **Test truth-hood**
  - `assertTrue(condition[, message])`
  - `assertFalse(condition[, message])`

- **Test values or objects for equality**
  - `assertEquals(expected, actual[, message])`
  - `assertNotEquals(expected, actual[, message])`

- **Test objects for identity (obj1 == obj2)**
  - `assertSame(expected, actual[, message])`
  - `assertNotSame(expected, actual[, message])`

- **Test null-hood**
  - `assertNull(object[, message])`
  - `assertNotNull(object[, message])`

- **Test exceptions**
  - `assertThrows(exception class, executable [, message])`

- **Automatic failure:** `fail(message)`
Recall the coding tip recommended.

- Idioms for testable code:
  - *Make message strings (and other values) constants*
  - *Make these members package-private (or public)*

- Example:

```java
public class GameCenter {

    public final static String NO_GAMES_MESSAGE = "No games have been played so far.";
    public final static String ONE_GAME_MESSAGE = "One game has been played so far.";
    public final static String GAMES_PLAYED_FORMAT = "There have been %d games played.";

    public synchronized String getGameStatsMessage() {
        if (totalGames > 1) {
            return String.format(GAMES_PLAYED_FORMAT, totalGames);
        } else if (totalGames == 1) {
            return ONE_GAME_MESSAGE;
        } else {
            return NO_GAMES_MESSAGE;
        }
    }
}
```
Here's how the test code would use these members to test multiple games being played.

```java
@Test
class GameCenterTest {
    @Test
    public void game_statistics_message_0() {
        // Setup the test scenario: no games play yet
        final GameCenter CuT = new GameCenter();
        // Analyze the results
        assertEquals(GameCenter.NO_GAMES_MESSAGE, CuT.getGameStatsMessage());
    }

    @Test
    public void game_statistics_message_1() {
        final GameCenter CuT = new GameCenter();
        // Simulate a single game ending
        CuT.gameFinished();
        // * Analyze the results
        assertEquals(GameCenter.ONE_GAME_MESSAGE, CuT.getGameStatsMessage());
    }

    @Test
    public void game_statistics_message_2() {
        final GameCenter CuT = new GameCenter();
        // Simulate two games ending
        CuT.gameFinished();
        CuT.gameFinished();
        // * Analyze the results
        assertEquals(String.format(GameCenter.GAMES_PLAYED_FORMAT, 2), CuT.getGameStatsMessage());
    }
}
```
When components have dependencies you have to consider how to isolate the dependencies.

- Dependencies are isolated along the testing seam for a component.

- There are three elements to consider
  - **Component under Test (CuT)**
  - **Friendly dependencies that can be trusted to work**
  - **Other dependencies that must have mocks because they are not trusted or we need special control during the test**
The UI routes are an interesting case because there are two components to test.

- The route itself creates the view map that the template engine uses
- The template engine receives the view map and uses the view template to generate the view HTML
- Each of those is a component to test.
- For example, for making guesses in the Guessing Game
These are the two testing configurations for making a guess.

PostGuessRoute Test

PostGuess Route

GAME CENTER
Request
Session
Player Services

Template Engine and Game View

TEST SEAM

Friendly

Game View Test

Template Engine and Game View

Mocks

Simulate the guess

Because the PostGuess route does not expose the view map, get access to it through Answer object.

Generate view map

Check the HTML
First, a quick review of dependency injection which is a key design technique to make classes testable.

```java
public class PostGuessRoute implements Route {

    private final GameCenter gameCenter;
    private final TemplateEngine templateEngine;

    /**
     * The constructor for the `POST /guess` route handler.
     *
     * @param gameCenter    The `GameCenter` for the application.
     * @param templateEngine template engine to use for rendering HTML pages
     *
     * @throws NullPointerException when the gameCenter parameter is null
     */
    PostGuessRoute(GameCenter gameCenter, TemplateEngine templateEngine) {
        // validation
        Objects.requireNonNull(gameCenter, "gameCenter must not be null");
        Objects.requireNonNull(templateEngine, "templateEngine must not be null");

        this.gameCenter = gameCenter;
        this.templateEngine = templateEngine;
    }
}
```

This class is instantiated by the `WebServer` component, which configures Spark.

This JDK method performs a null check and throws an NPE if violated.

The `WebServer` component injects these two dependencies into this route.
Some classes are thoroughly tested and can be considered friendly; others we must mock.

```java
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.ArgumentMatchers.eq;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;

@Tag("UI-tier")
public class PostGuessRouteTest {

    private static final int NUMBER = 7;
    private static final int WRONG_GUESS = 3;
    private static final String WRONG_GUESS_STR = Integer.toString(WRONG_GUESS);
    private static final String NOT_A_NUMBER = "asdf";
    private static final int INVALID_GUESS = 47;
    private static final String INVALID_GUESS_STR = Integer.toString(INVALID_GUESS);

    /**
     * The component-under-test (CuT).
     */
    private PostGuessRoute CuT;

    // friendly objects
    private GameCenter gameCenter;
    private GuessGame game;

    // attributes holding mock objects
    private Request request;
    private Session session;
    private TemplateEngine engine;
    private PlayerServices playerSvc;

    Some classes are thoroughly tested and can be considered friendly; others we must mock.

    GameCenter and Game are tested so we consider them friendly.

    These we need to mock to isolate the unit under test.
```
We have to setup the mocks before each test.

```java
@BeforeEach
public void setup() {
    request = mock(Request.class);
    session = mock(Session.class);
    when(request.session()).thenReturn(session);
    engine = mock(TemplateEngine.class);

    // build the Service and Model objects
    // the GameCenter and GuessingGame are friendly
    gameCenter = new GameCenter();
    game = new GuessGame(NUMBER);

    // but mock up the PlayerService
    playerSvc = mock(PlayerServices.class);
    when(playerSvc.currentGame()).thenReturn(game);

    // store in the Session
    when(session.attribute(GetHomeRoute.PLAYERSERVICES_KEY)).thenReturn(playerSvc);

    // create a unique CuT for each test
    CuT = new PostGuessRoute(gameCenter, engine);
}
```

Use the Mockito `mock` function to create a mock object

Use the Mockito `when` and `thenReturn` APIs to simulate a method call on a mock object.

Inject the setup mocks into the CuT.
We set up to test the PostGuessRoute including access to data on the Route-TemplateEngine seam.

- Mockito's Answer mechanism allows us to save, and later analyze intermediate data crossing the seam.

```java
@Test
public void bad_guess_1() {
    // Arrange the test scenario: The user's guess is not valid number.
    when(request.queryParams(eq(PostGuessRoute.GUESS_PARAM))).thenReturn(NOT_A_NUMBER);

    // To analyze what the Route created in the View-Model map you need
    // to be able to extract the argument to the TemplateEngine.render method.
    // Mock up the 'render' method by supplying a Mockito 'Answer' object
    // that captures the ModelAndView data passed to the template engine
    final TemplateEngineTester testHelper = new TemplateEngineTester();
    when(engine.render(any(ModelAndView.class))).thenAnswer(testHelper.makeAnswer());
}
```

For this specific test case, we also need to specify the Request 'guess' query parameter.

TemplateEngineTester is a provided helper class for testing the ModelAndView passed to TemplateEngine.render().
With the mocks setup, now direct the scenario of the test.

```java
@Test
class PostGuessRouteTest {

    @Test
    public void bad_guess_1() {
        ...

        // Invoke the test
        CuT.handle(request, response);

        // Analyze the results:
        //   * model is a non-null Map
        testHelper.assertViewModelExists();
        testHelper.assertViewModelIsaMap();
        //   * model contains all necessary View-Model data
        testHelper.assertViewModelAttribute(GetHomeRoute.TITLE_ATTR, GetGameRoute.TITLE);
        testHelper.assertViewModelAttribute(GetHomeRoute.NEW_PLAYER_ATTR, Boolean.FALSE);
        testHelper.assertViewModelAttribute(PostGuessRoute.MESSAGE_TYPE_ATTR, PostGuessRoute.ERROR_TYPE);
        testHelper.assertViewModelAttribute(PostGuessRoute.MESSAGE_ATTR, PostGuessRoute.makeBadArgMessage(NOT_A_NUMBER));
        testHelper.assertViewModelAttributeIsAbsent(PostGuessRoute.YOU_WON_ATTR);
        //   * test view name
        testHelper.assertViewName(GetGameRoute.VIEW_NAME);
    }
}
```

For this test of the route itself, we do not care about the HTML generated by the call to `handle()`.
There will also be a suite of unit tests to check if the view generates the correct HTML.

```java
@Tag("UI-tier")
public class GameViewTest {

    private static final String TITLE = "MyTitle";
    private static final String TITLE_HEAD_TAG = "<title>" + TITLE + "</title>";
    private static final String TITLE_H1_TAG = "<h1>" + TITLE + "</h1>";

    private static final String GAME_HEADING_1ST_TAG = "<h4>Make a Guess</h4>";
    private static final String GAME_HEADING_TAG = "<h4>Make Another Guess</h4>";

    private static final String BAD_GUESS_MSG = "Bad guess.";

    /**
     * Make the HTML text for a div element that holds a user message.
     */
    private static String makeMessageTag(final String text, final String type) {
        return String.format("<div class=\"message %s\">%s</div>\", type, text);
    }

    /**
     * Make the message text for how many guesses are left.
     */
    private static String makeGuessesLeftMsg(final int guessesLeft) {
        if (guessesLeft > 1) {
            return String.format("You have %d guesses left.", guessesLeft);
        } else {
            return "You have 1 guess left.";
        }
    }

    private final TemplateEngine engine = new FreeMarkerEngine();
```

This is what we expect in the generated HTML.

Some helper methods to create other expected HTML strings.

The component under test.
The test will check the rendered HTML for one of the view scenarios.

```java
@Tag("UI-tier")
public class GameViewTest {

    // Create the view map for this test scenario.
    public void bad_data_message() {
        // Arrange test
        final Map<String, Object> vm = new HashMap<>();
        final ModelAndView modelAndView = new ModelAndView(vm, GetGameRoute.VIEW_NAME);
        // setup View-Model for a new player
        vm.put(GetHomeRoute.TITLE_ATTR, TITLE);
        vm.put(GetGameRoute.GAME_BEGINS_ATTR, false);
        vm.put(PostGuessRoute.MESSAGE_ATTR, BAD_GUESS_MSG);
        vm.put(PostGuessRoute.MESSAGE_TYPE_ATTR, PostGuessRoute.ERROR_TYPE);
        vm.put(GetGameRoute.GUESSES_LEFT_ATTR, 2);

        // Invoke test
        final String viewHtml = engine.render(modelAndView);

        // Generate the view HTML.
        // Analyze results
        // * look for Title elements
        assertTrue(viewHtml.contains(TITLE_HEAD_TAG), "Title head tag exists.");
        assertTrue(viewHtml.contains(TITLE_H1_TAG), "Title heading tag exists.");
        // * look for the Game heading
        assertTrue(viewHtml.contains(GAME_HEADING_TAG), "The Game heading tag exists.");
        // * look for the error message
        assertTrue(viewHtml.contains(makeMessageTag(BAD_GUESS_MSG, PostGuessRoute.ERROR_TYPE)),
                    "Error message tag exists.");
        // * look for the 'Guesses left' message
        assertTrue(viewHtml.contains(makeGuessesLeftMsg(2)), "Guesses left message exists.");
    }
}
```

Verify that the HTML has the strings that we expect.
Mockito has a rich API for setting scenarios and for inspecting test activity.

- Arranging scenarios:
  - `when(mock.method(args)).thenReturn(value)`
  - `when(mock.method(args)).thenThrow(new XyzExc())`
  - `when(mock.method(args)).thenAnswer(lambda)`

- Inspecting activity within the CuT method:
  - `verify(mock).method(args)`
  - `verify(mock, times(1)).method(args)`

- Other verification modes:
  - `times(n), atLeast(n), atMost(n) & never()`

- Specifying arguments:
  - *An actual value or object*: `eq(value)` or `eq(object)`
  - *Any value* (`anyInt()`, etc); `any()` for any *Object*
  - Many more types of **Matchers**
Your exercise is to build unit tests for two classes in your Project.

✓ Each team member picks two classes in your project and builds unit tests for them.
  • Each team member will pick different classes to test.
  • If you need to refactor the code to make the component more testable then do so.

✓ Create the test class in the appropriate package in the test source path.

✓ Create a reasonable set of test cases.
  • At least three test cases for each CuT.
  • Focus on the business logic methods (eg, the handle method for UI Controllers).
Your exercise is to build unit tests for two classes in your Project.

✓Upload the two unit test source files to the Unit testing - individual Dropbox in MyCourses.

✓You will now complete the Definition of Done unit testing checklist items to consider the story done.